RAPPORT

Nom : Hiba BACHA
Langage : Java

1. Objectif du projet

Le but de ce projet était de créer un système capable de **compresser des tableaux d'entiers** pour réduire leur taille, tout en gardant la possibilité d'accéder directement à un élément sans avoir à tout décompresser.
En d'autres termes, on voulait trouver un **compromis entre compacité et rapidité**.

Le sujet demandait d'implémenter **trois versions** d'un encodeur :

1. **Sans chevauchement**, où chaque valeur est stockée dans un mot de 32 bits sans dépasser la limite.
2. **Avec chevauchement**, où les valeurs peuvent se couper entre deux mots pour ne perdre aucun bit.
3. **Avec débordement (overflow)**, où seules les petites valeurs sont compressées normalement et les valeurs trop grandes sont placées dans une zone spéciale à la fin.

Le projet devait aussi inclure :

- des **mesures de performance** (temps de compression, décompression, accès direct),
- et une **gestion des entiers négatifs**, qui n'était pas demandée explicitement mais que j'ai ajoutée pour rendre le programme plus complet.

2. Structure du projet et fonctionnement

J'ai développé ce projet en **Java**, car c'est un langage stable et bien adapté à la manipulation d'entiers et de bits.
Le projet est découpé en plusieurs classes, chacune ayant un rôle précis.

Interface CodeurBits : C'est l'interface principale du projet.
Elle définit les trois fonctions essentielles que toutes les versions doivent avoir :

```
int[] compresser(int[] tableau);

int[] decompresser(int[] tableau);

int acceder(int index);
```

Grâce à cette interface, chaque version (sans chevauchement, avec chevauchement ou avec débordement) suit la même structure.

b) CodeurSansChev

Cette classe réalise la compression **sans chevauchement**.
Chaque entier est codé sur k bits, et plusieurs valeurs sont regroupées dans un seul mot de 32 bits.

C’est la version la plus simple à comprendre et à déboguer.
Elle perd un peu de place à la fin des mots (quelques bits inutilisés), mais elle est très rapide et stable.

c) CodeurAvecChev

Ici, les entiers sont écrits les uns à la suite des autres dans un flux continu de bits, sans tenir compte des frontières de mots.
Une valeur peut donc être partagée entre deux mots.
C’est plus compact, mais aussi plus délicat à gérer, car il faut manipuler les décalages (<<, >>>) avec précision.
J’ai utilisé les fonctions de la classe OutilsBits pour fiabiliser cette partie.

d) CodeurDebordement

Cette version ajoute une **zone overflow** pour les valeurs trop grandes.
Chaque entier est encodé sur kBase bits, avec un **bit de tag** :

- 0 = valeur directe,
- 1 = index vers la zone overflow.

Les grandes valeurs sont donc stockées séparément en fin de tableau.
Cette approche permet d’économiser beaucoup d’espace quand il y a seulement quelques “grands” nombres dans un tableau.

e) OutilsBits

C’est une classe utilitaire qui contient les fonctions bas niveau pour écrire et lire des bits dans les tableaux d’entiers :

- ecrireBits(int[] buf, long pos, int nbits, int valeur)
- lireBits(int[] buf, long pos, int nbits)
- et quelques fonctions d’aide comme masque() ou bitsNecessaires().

Centraliser ces fonctions a évité beaucoup d’erreurs de décalage et rendu le code plus clair.

f) FactoryCodeurBits

Cette classe permet de créer facilement le bon codeur en fonction du mode demandé :

FactoryCodeurBits.creer("sans");

FactoryCodeurBits.creer("avec");

FactoryCodeurBits.creer("debordement");

C’est pratique pour tester rapidement chaque version sans changer tout le code principal.

g) BenchProto

Cette classe sert à mesurer les temps d’exécution de chaque opération.
Les tests se font sur plusieurs répétitions pour obtenir une moyenne fiable.
J’ai utilisé System.nanoTime() pour mesurer les temps en millisecondes.
Les résultats sont ensuite affichés sous la forme :

Mesures (ms) : {compresser_ms=0.0017, decompresser_ms=0.0010, acceder_ms=0.0003}

Cela permet de comparer les performances des trois versions.

h) DemoEnonce

C'est la classe principale qui montre le fonctionnement du projet.
Elle effectue plusieurs tests :

- un tableau simple avec des négatifs [-5, -1, 0, 3, 7],
- un tableau aléatoire avec des valeurs mixtes (positives et négatives),
- et l'exemple officiel de l'énoncé pour la version overflow : [1, 2, 3, 1024, 4, 5, 2048].

Le programme vérifie que :

- get(i) renvoie la bonne valeur,
- decompresser(compresser(A)) == A,
- et affiche ok=true si tout est correct.

i) Main

Cette classe sert juste de point d'entrée pour lancer le projet avec la commande :

java -cp out application.DemoEnonce sans

ou avec avec ou debordement selon la version à tester.

3. Choix techniques et solutions

- **Langage Java** : choisi pour sa stabilité, sa gestion native des entiers et la clarté de sa syntaxe.
- **Architecture modulaire** : interface + 3 implémentations + fabrique. Cela rend le code organisé, facile à tester et à faire évoluer.
- **Gestion des entiers négatifs** : ajout d'un **offset** (décalage des valeurs) avant compression, retiré après décompression. Simple et efficace.
- **Mesures de performance** : réalisées sur plusieurs répétitions pour compenser les variations de la JVM.
- **Validation** : tous les tests donnent ok=true, preuve que la compression et la décompression sont sans perte.

4. Difficultés rencontrées et comment je les ai résolues

1. **Manipulation des bits** :
   Au début, je faisais souvent des erreurs de décalage ou de masque, ce qui corrompait les données.
   J'ai corrigé ça en isolant toute la logique binaire dans OutilsBits, et en testant chaque fonction indépendamment avant de les réutiliser.
2. **Accès direct (get(i))** :
   Calculer la position exacte d'un élément dans le flux compressé n'était pas évident, surtout

en mode chevauchement.
J’ai ajouté des formules précises et vérifié les résultats sur des tableaux très courts avant de généraliser.

3. **Débordement (overflow)** :
   Le plus dur a été de trouver comment choisir la bonne valeur de kBase pour minimiser la taille totale du buffer.
   J’ai implémenté une boucle qui teste plusieurs valeurs et garde celle qui donne la meilleure taille finale.

4. **Tests et validation** :
   J’ai créé une démo complète (DemoEnonce) qui exécute automatiquement les trois modes et compare les résultats.
   Cela m’a aidée à repérer rapidement les erreurs et à valider que tout fonctionne.

5. **Négatifs** :
   Comme ce n’était pas prévu dans l’énoncé, j’ai cherché une méthode simple.
   L’offset est devenu la meilleure solution, car elle garde le code cohérent sans changer la logique générale.

5. Conclusion

La version sans chevauchement est la plus simple à comprendre,
celle avec chevauchement la plus compacte,
et la version overflow la plus flexible pour les données hétérogènes.

Ce projet m’a appris à manipuler les bits avec précision, à structurer un programme de façon modulaire et à tester chaque partie séparément avant de tout assembler